CGI2012 manuscript No.
(will be inserted by the editor)

Patch-Based Occlusion Culling for Hardware Tessellation

Matthias Nießner · Charles Loop

Abstract We present an occlusion culling algorithm that leverages the unique characteristics of patch primitives within the hardware tessellation pipeline. That is, unseen patches are costly to compute, but easy to find screen space bounds for. Our algorithm uses the well known hierarchical Z-buffer approach, but we present a novel variant that uses temporal coherence to maintain lists of visible and occluded patches. Patches may be animated and have applied displacement maps. We also allow traditional triangle mesh geometry to serve as occluders. This makes our scheme ideally suited for patch based articulated character models, moving within a polygonal environment.

Keywords Real-time Rendering · Hardware Tessellation · Culling

1 Introduction

The recent elevation of patch primitives to first class objects in the graphics pipeline offers a unique opportunity to revisit classic approaches to visibility culling and improve upon them with new insights tailored to the computational demands of patch processing. Previous occlusion culling algorithms required a scene graph structure whose leaves bound a sufficiently dense, and static collection of geometry in order amortize their cost. This limited occlusion culling to fixed scene graphs that must be traversed at runtime. Our new patch based approach is *unstructured* in the sense that we can process standard lists (buffers) of patch primitives and still achieve significant performance gains.

The key observations we leverage are 1) that patches are compact and are (relatively) easy to find bounds for in screen space; and 2) that committing to the evaluation, tessellation, and generation and rejection of triangles, corresponding to a patch is (relatively) expensive. Since the cost of processing patches that are not seen is high, and the cost of finding screen space bounds for patches is low, the extra computation needed to perform occlusion culling is easily amortized. The algorithm we propose in this paper is easy to implement and requires no pre-processing of patch based models. Since the screen space bounds of individual patches are re-computed every frame, we automatically support animated models. Furthermore, to maximize utility, we allow patches to have displacement maps applied and offer a novel technique for bounding such patches.

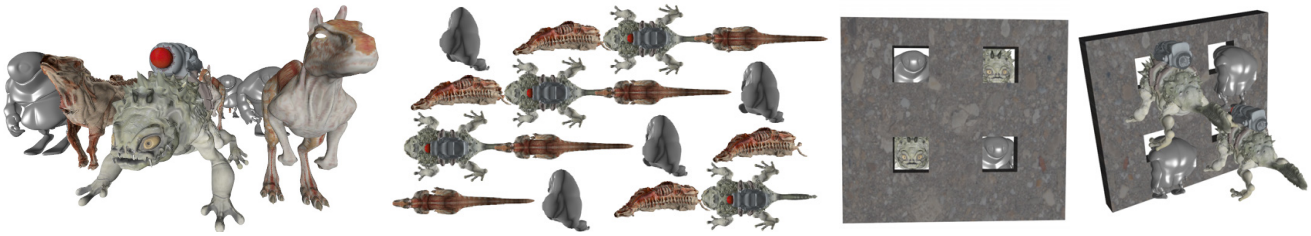
The strategy behind our algorithm is to use temporal coherence to maintain the visible/occluded status of individual patches, use the visible patches to build a hierarchical Z-buffer [4] on-the-fly, and then to update the visibility status of patches. Our method is conservative in that occluded patches may occasionally be rendered (needlessly), but visible patches are always rendered. The culling overhead for our algorithm is small compared to the computational savings, resulting in significant performance gains (see Figure 1). Note that even for simple scenes (e.g., single objects) and small tess factors our algorithm is effective (see Figure 9).

We summarize our main contributions as follows:

- Fast and efficient occlusion culling for patches
- Novel bounds for patches with displacements
- Dynamic scenes with patch occluders and occludees

M. Nießner
University of Erlangen-Nuremberg
E-mail: matthias.niessner@cs.fau.de

C. Loop
Microsoft Research
E-mail: charles.loop@microsoft.com



(a) 5.7 vs 12.1 ms (b) 9.1 vs 11.1 ms (c) 3.4 vs 9.0 ms (d) 6.7 vs 8.7 ms

Fig. 1 Our algorithm performs culling on a per patch basis and supports patches w/ and w/o displacements. The images above are rendered both using, and not using, our culling method; see the performance gain below the respective image. Our method significantly speeds up rendering of scenes with even moderate depth complexity (a, 64.2% culled) including triangle mesh occluders (c, 70.6% culled). Even when having intra-object occlusion only, render time can be reduced (b, 30.4% culled; d, 30.5% culled).

2 Previous Work

Hardware Tessellation: Hardware tessellation allows parametric surface patches to be evaluated and rendered on-chip, greatly reducing costly off-chip memory accesses. Several patching schemes have been proposed that are accelerated by this architecture [16], [7], [11]. Our work is agnostic to a particular patching scheme; our only assumption is that patches obey the convex-hull property, and the first partial derivatives can be bounded. For simplicity, we consider widely used bicubic Bézier patches, though our algorithm is easily extended to other patch types.

Back-patch Culling Techniques: While *back-face* culling is a standard method in today's graphics hardware to reduce triangle rasterization and pixel/fragment shading operations, *back-patch* culling can be analogously applied in order to avoid costly surface evaluations. Such schemes take patch normals or tangent planes into account in order to determine whether a patch is back or front-facing. The *cone of normals* is one such technique [14]. Though the cone of normals provides tight normal bounds, its computation is relatively costly. An approximate cone of normals can be computed at low cost by combining a tangent and bitangent cone [12]. Munkberg et al. [10] used this in the context of bounding displaced patches. We use a similar idea to deal with displaced patches, however, our bounds are optimized with respect to screen space area and thus provide better culling results. A general problem with back-patch culling is the inability to deal with displaced patches. Hasselgren et al. [6] address this by using a Taylor series to represent the displaced surface. Nevertheless, this has severe limitations (e.g., cannot deal with fractional tessellation) and its cull rate is poor. A near optimal back-patch culling technique, making use of a parametric tangent plane, has recently appeared [8]. Though effective, the cost of this method is relatively high.

Occlusion Culling: Instead of following a back-patch culling approach, our method considers occlusion culling of patches. There exist many occlusion algorithms in the context of polygon rendering. A survey of early methods is provided by Cohen-or et al. [2]. On modern GPUs, hardware occlusion queries provide information about whether an object contributes to the current frame. In OpenGL `GL_NV_conditional_render` (predicate rendering in D3D) allows conditional rendering without GPU-CPU synchronization; i.e., if an occlusion query is not yet complete until the next conditional draw call, rendering will be performed ignoring the actual query result. There are several methods that efficently use hardware occlusion queries by reducing the number of issued queries in the context of per object occlusion culling [13], [1], [5], [9]. However, all these algorithms share several problems that makes their application on a per patch level inefficient: they require separate draw calls for each cull primitive (this severely affects performance since thousands of separate draw calls could be necessary to render a single object); spatial hierarchies are required on the CPU side to limit the number of issued queries (updates become costly under animation); rasterizing bounding geometry creates additional overhead (latency and compute). Engelhardt and Dachsbacher [3] propose two methods for granular visibility queries that make query results available on the GPU: pixel counting with summed area tables and hierarchical item buffers. The first method assigns query objects to color channels and uses summed area tables to count covered pixels. Thus, only four query objects per region are supported, which is insufficent considering thousands of patches whose bounds overlap particularly when considering displacements. Hierarchical item buffers write IDs of query objects to resulting pixels. The resulting buffer is interpreted as a point list and in a second (count) renderpass a vertex shader distributes points (i.e., query IDs) to pixels accordingly. With alpha blending enabled the number of covered

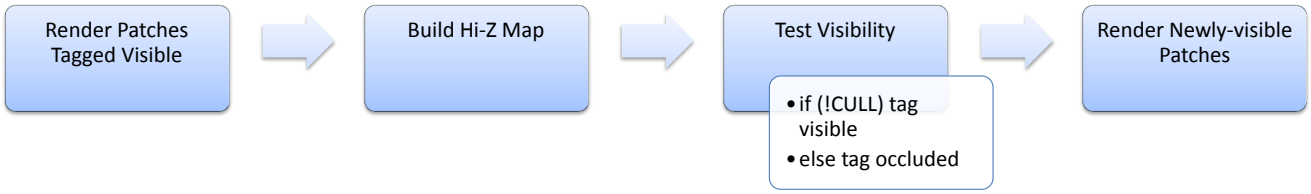


Fig. 2 Overview of our culling pipeline within a frame: first, patches tagged visible are rendered; the resulting depth buffer is used to construct the Hi-Z map. Next, all patches are tested for visibility against the Hi-Z map. Last, all patches that were previously tagged occluded but are now visible (i.e., newly-visible) are rendered.

pixels can be obtained for each query object. While this might be possible for a larger number of query objects, rendering a single point for each pixel (i.e., in practice over a million points) seems to be significant overhead. In addition, on-the-fly bounding geometry computation and rasterization remains a problem.

We base our method on the Hierarchical Z-buffer proposed by Greene et al. [4]. They use an object-space octree of the scene geometry and a screen space Z-pyramid (Hi-Z map). The pyramid's lowest level is the Z-buffer; each higher level is constructed by combining four Z values into a Z value at the next lower level by choosing the farthest Z value from the observer. Then the cubes of the octree are tested against the best fitting entry in the Hi-Z map. Shopf et al. [15] use the Hi-Z map to perform culling based on geometry instances. As an extension to the original hierarchical Z approach, they use bounding spheres and four Hi-Z map samples to obtain better coverage. However, in their approach occluders (i.e., the terrain) are fixed and occlusion culling is only applied to selected objects (i.e., characters). Since they use geometry instances tested objects all (must) have exactly the same topology and obtaining the cull decisions involves a CPU query.

In contrast to previous occlusion culling methods our approach supports fully animated objects and does not require any pre-computed scene data structures. Furthermore, we are able to cull subsets of objects since our algorithm works at the patch level; there are no static occluder lists, so all objects can act as occluders or be occluded.

3 Culling Pipeline

Our algorithm works by maintaining visibility status bits of individual patches (visible, occluded, or newly-visible) as each frame is rendered. Assume that a frame has already been rendered and these status bits have been assigned to patches. At the start of a new frame, patches marked as visible are rendered. From the Z-buffer of this frame, a hierarchical Z-buffer (or *Hi-Z map*) is built using a compute shader (or CUDA kernel). Next, all patches are occlusion tested against the newly-constructed Hi-Z map. Patches passing this test (i.e., not occluded) are either marked as visible if they were previously visible, or newly-visible if they were previously occluded; otherwise they are marked as occluded. Finally, all patches marked as newly-visible are rendered to complete the frame. See Figure 2 for a flow diagram of this process.

The simplest way to initialize the visibility status bits of patches would be to mark all patches visible. However, this would cause all patches to be rendered in the first frame. To avoid this worst-case behavior, we mark half of the patches as visible and the other half occluded. Even randomly assigning visibility status bits will allow at least some patches to be occluded; which is better than none. After the first frame has been rendered, however, we rely on temporal coherence to approximate these visible and occluded sets. Our observation is that between frames, most visible patches stay visible while most occluded patches stay occluded. Obviously, for dynamic scenes some visible patches will become occluded between frames, and vice versa (see Figure 9 right). One of the key features of our algorithm is to efficiently track these changes so that each new frame begins with a good approximation to the set of visible patches.

While the main focus of our work is on hardware tessellation of animated, patch-based objects such as characters, scene environments are often represented by triangle meshes. We render triangle meshes before generating the Hi-Z map so their depth information is included in the Hi-Z map construction. This will allow triangle meshes to be treated as occluders for patch-based objects. It is also straightforward to combine this per patch algorithm with previous approaches that focus on entire objects. The Hi-Z map information can be used to determine whether an object's bounding volume is fully occluded. However, computing bounding volumes of dynamic objects on-the-fly can be be costly.

3.1 Aggressive Culling

The pipeline shown in Figure 3 can be even simplified by omitting the last render pass. Updating patch tags will still fix rendering in the subsequent frame. This will cause patches becoming visible to appear with a one frame delay. This may be tolerable in some real-time applications since that delay may not be noticeable at high frame-rates. However, we do not evaluate this option in our results.

4 Applying Cull Decision

We now describe how to determine patch visibility within the culling pipeline presented in Section 3. We separate this into the problem of obtaining occlusion information (i.e., Hi-Z map creation) and the culling test itself.

4.1 Computing Occlusion Data

As mentioned in Section 3, visible patches (i.e., occluders) are rendered first. In order to obtain occlusion information we employ a hierarchical Z-buffer approach. Therefore, we use the depth buffer resulting from rendering the visible patches to generate a Hi-Z map [4], [15].

The Hi-Z map construction is similar to standard mip mapping where four texels are combined to determine a single texel of the next mip level. Instead of averaging texels, the value of the new texel is obtained by the maximum depth value of the corresponding four child texels (i.e., it is set to the largest distance value). Thus, within a region covered by a particular texel (no matter which mip level) a conservative bound is given, so that at the texel's location no objects with a larger depth value are visible. Five levels of an example Hi-Z map are shown in Figure 3. Note that watertight patch joins are crucial for a good Hi-Z map since cracks at patch boundaries will propagate a false depth value through the hierarchy.

We found binding the hardware depth buffer to be relatively costly, see Figure 7 right. Though the highest resolution level of the Hi-Z map could correspond to depth buffer, we avoid copying this data and use a half resolution image as the highest level. This is reasonable as the full resolution map would only be useful to cull tiny patches. We assume that such tiny patches will not be tessellated with a high enough tess factor to make patch occlusion culling effective; thus the highest level is not needed.

The different Hi-Z levels are stored in a single texture with its respective mip levels to obtain fast access. We deal with non-power-of-two size images by enlarging the Hi-Z map's width and height to the next greater (or equal) power of two. This is necessary since the default size of mipmap levels will always be even; e.g., a 5×5 texture will be down sampled to 2×2. The resized Hi-Z map, however, allows us to take all texels of the parent level into account; e.g., a 5×5 texture will be down sampled to 3×3. While unused texels are initialized with 0, all kernels can be used without any modification.

Fig. 3 Five levels (0,4,5,6,7) of a Hi-Z map corresponding to a view of the Bigguy model.

4.1.1 Cull Decision

Cull decisions are applied per patch. As a representative patch primitive we use bicubic Bézier patches consisting of a 4×4 array of control points (the basic approach could be applied to various other patching schemes). For now we assume the patches do not have displacement maps applied; we will extend the algorithm to include displacements in Section 4.2. In order to determine visibility of a patch we compute its axis-aligned bounding box (AABB) in clip space. We use the AABB's front plane to test against the Hi-Z map (see Section 4.1). Depending on the bounding box's width and height in screen space a particular level of the Hi-Z map is chosen: $level = \lceil \log_2(\max(width, height)) \rceil$. The bounding box's area will be conservatively covered by at most 4 texels of the selected Hi-Z map level. Considering multiple Hi-Z map entries allows us to achieve better coverage; see Figure 4 for the distinct Hi-Z access patterns. If the respective depth values of the Hi-Z map are all smaller than the minimum Z value of the patch's bounding box, we set the visibility status bit of the patch to occluded.

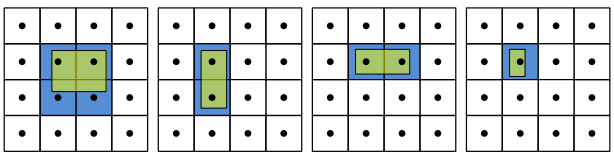


Fig. 4 Different Hi-Z access patterns; at most four texels (blue) are chosen to cover the screen space bounding rectangle (yellow) of a patch. The case that only one texel is used (right) is special since it only applies at the finest Hi-Z level where no further refinement is possible.

Since we must compute the screen space bounding box of a patch to perform occlusion culling, we can also apply view frustum culling at virtually no cost. That is, patches whose bounding box lies entirely outside of the clipping cube are culled. We clamp bounding box corners to screen space if they are partially outside, to ensure that all corners map to Hi-Z values. While we perform both culling methods in the same kernel, view frustum culling is applied first since its computation is less costly.

4.2 Displaced Patches

Displacement mapping is an important use case for hardware tessellation as it allows adding high resolution geometric detail at low cost. We now adapt our algorithm to handle patches with displacements. Although determining the patch bounds is different, creating the Hi-Z map (see Section 4.1) is the same for patches with, and without, displacements since we obtain visibility information directly from the depth buffer.

For displaced patches, we use a camera-aligned frustum (CAF) as a bounding volume; that is, a frustum whose side planes contain the origin in camera space. In camera space the eyepoint is at the origin and the viewing direction is the positive z-axis (in OpenGL it's the negative z-axis). Unlike the non-displaced case, we do not yet apply the perspective transform since this projective mapping would destroy Euclidean distance metrics that we rely on to construct our bounds.

First, control points are transformed to camera space. We then find the minimum Z value ($minZ$) among the control points to determine the front plane of the CAF (a plane perpendicular to the viewing direction). Next, we project the patch control points onto the CAF's front plane. Since control points P_i are in camera space, we can achieve this by coordinate rescaling: $P'_i = P_i \cdot \frac{minZ}{P_{i,z}}$. Side planes of the CAF are then obtained by computing the minimum and maximum x, y values of the P'_i. Note that the resulting frustum is the generalization of the screen space AABB. Therefore, its screen space projection will give the same result non-displaced patches (Section 4.1.1), if all displacements were zero.

In order to bound the range of patch normals to which displacements are applied, we compute a cone-of-normals for each patch. This cone is represented by an aperture angle α and cone axis $\mathbf{a}$. We consider the construction of Shirmun and Abi-Ezzi [14] (accurate cone) or the method of Sederberg and Meyers [12] (approximate cone); the trade-off being accuracy versus computational cost. Since obtaining the accurate cone is relatively costly (see Figure 7 left), our choice for animated patches will be the approximate version. The accurate cone may be still be used for static objects where the cone can be precomputed.

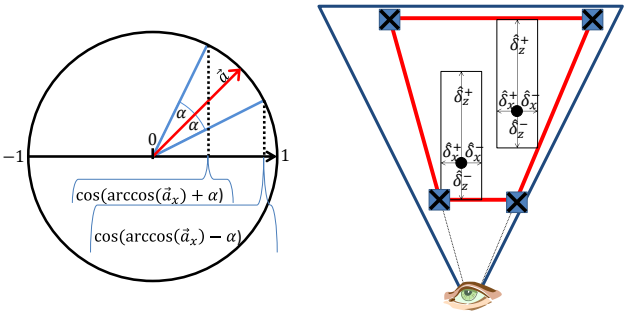


Fig. 5 The construction of our camera-aligned frustum (CAF). Left; the cone axis $\mathbf{a}$ and the aperture α determine the extension in the positive x direction (δ_x^+). Right; the CAF (red) is determined in camera space for two control points.

We also find bounds on the scalar displacement values and require the maximum $D_{\max}$ to be positive (or zero) and the minimum $D_{\min}$ be negative (or zero). The displacement bounds, the cone axis $\mathbf{a}$, and aperture α, are used to extend the CAF so that it will conservatively bound the displaced patch. For each camera space coordinate, we compute positive and negative extensions as follows (see Figure 5 left):

$$
\begin{aligned}
&\textit{if}(\mathbf{a}_x \geq \cos(\alpha)) \\
&\quad \delta_x^+ = 1 \\
&\textit{else} \\
&\quad \delta_x^+ = \max(\cos(\arccos(\mathbf{a}_x) + \alpha), \cos(\arccos(\mathbf{a}_x) - \alpha))
\end{aligned}
$$

$$
\begin{aligned}
&\textit{if}(-\mathbf{a}_x \geq \cos(\alpha)) \\
&\quad \delta_x^- = 1 \\
&\textit{else} \\
&\quad \delta_x^- = \max(\cos(\arccos(-\mathbf{a}_x) + \alpha), \cos(\arccos(-\mathbf{a}_x) - \alpha))
\end{aligned}
$$

This allows us to compute a minimum and maximum bound for each control point:

$$
\begin{aligned}
P_{\max} &= P + \hat{\delta}_x^+ = P + \max(D_{\max} \cdot \delta^+, -D_{\min} \cdot \delta^-) \\
P_{\min} &= P - \hat{\delta}_x^- = P - \max(D_{\max} \cdot \delta^-, -D_{\min} \cdot \delta^+)
\end{aligned}
$$

$P_{\max}$ and $P_{\min}$ define AABBs for each control point and determine the $minZ$ value, i.e., the CAF front plane. In order to construct the CAF, the corner points of all AABBs are projected on the CAF front plane (see Figure 5, right). Finally, the CAF is transformed into screen space by projecting the four corner points of its front plane. Visibility for the screen space bounding box is determined the same way as described in Section 4.1.1. Both occlusion and view frustum culling benefit from the tight CAF bounds.

Note that for a given cone of normals the CAF provides an optimal screen space bound. Figure 6 shows the difference between our bounds and state-of-the-art previous work using both the same cone.

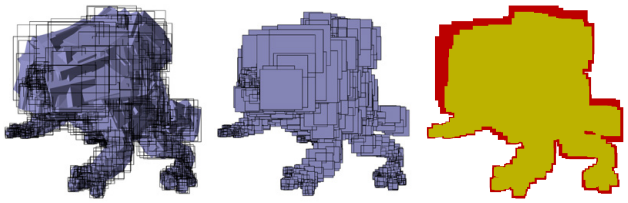


Fig. 6 Different bounding methods for displaced surfaces visualizing object (purple) and screen space (black quads) bounds: OOBB (object-oriented bounding box [10]), CAF (ours) and a comparison between OOBB (red) and CAF (yellow); both using the approximate cone of normals. Our bounds generalize axis-aligned screen space bounds and thus provide optimal results for a given cone of normals.

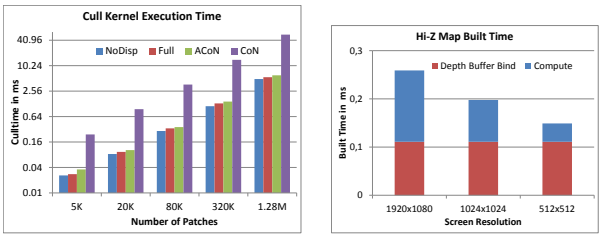


Fig. 7 Left: execution times for different culling kernels with varying patch count. Having no displacements (NoDisp), uniform bounding volume extension (Full), and considering approximate cone of normals (ACoN) takes about the same amount of time, while using the true cone (CoN) is significantly more expensive. Right: Hi-Z map construction times for different screen resolutions. Timings are split by binding the depth buffer (depth stencil view) and running the Hi-Z build kernel.

5 Implementation Details

Our algorithm was implemented using DirectX 11 with HLSL shaders. Since patches are tagged either *visible* or *occluded*, a binary flag is sufficient to define visibility status. A second binary flag is required to mark a patch as *newly-visible*, to be rendered in the next draw call. Both flags are combined in a per object texture that contains one value for each patch. Cull decisions are determined in a compute shader that runs one thread per patch, with the results being stored in the previously mentioned texture. This texture is then accessed in the constant hull shader, where patch culling is applied by setting respective tess factors to zero. This turned out to be faster than computing cull decisions directly in the hull shader. We attribute the poor constant hull shader performance to the fact that there is only a single constant hull shader thread running per warp. Since there is no measurable context switch overhead between rendering and compute in DirectX 11, we also use compute shaders to construct the Hi-Z map.

6 Results

All experimental results were made using an NVIDIA GeForce GTX 480. Timings are provided in milliseconds and account for all runtime overhead except for display of the GUI widgets, text rendering, etc.. In order to reflect a real application use case, backface culling is always turned on to explicitly measure computations done by hardware tessellation and costs associated with front facing fragments.

6.1 Cull Computations

Our algorithm requires passes for drawing, determining visibility, and Hi-Z map construction. Figure 7 left shows the runtime of the culling kernels for different patch counts. Culling time scales linearly with respect to the number of patches. The non-displaced surface (NoDisp), the simple uniform frustum extension (Full) and the approximate cone of normals (ACoN) kernel are approximately the same cost, while the accurate cone of normals bounding frustum extension kernel (CoN) is about an order of magnitude slower. These results suggest that the ACoN kernel is the best choice for dynamic displaced surfaces, since its slightly lower effectiveness compared to the CoN kernel (see Section 6.2), is offset by its significantly lower cost. In fact the CoN kernel is not suitable for animated patches since rendering the patches is cheaper in most cases than the cull kernel execution. The corresponding kernels using the bounds of Munkerberg et al. [10] have about the same cost ($\pm 3\%$), however, their cull-rate is always lower (see Table 1).

The performance of the Hi-Z map construction for various screen resolutions is shown in Figure 7 right (the depth buffer is down sampled to a size of 4×4 pixels). A large portion the Hi-Z map computation time is consumed binding the current depth buffer. Our speculation is that binding the depth buffer causes a copy operation due to the driver/vendor specific internal depth buffer representation. Further, the Hi-Z map creation is independent of the number of patches and is therefore a constant factor in our culling pipeline.

While our method executes both cull and Hi-Z map construction kernels, the time required by our algorithm for a scene with 80K patches was less than a millisecond. Depending on the cull rates we expect a pay off even at low surface evaluation costs (see Section 6.2, 6.3).

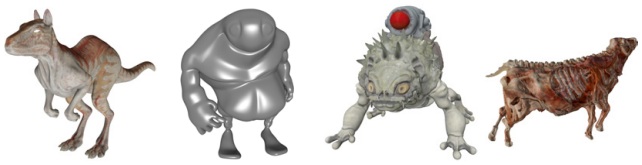


Fig. 8 Test models: Killeroo and Bigguy (non-displaced; 2.9K and 1.3K patches); Monsterfrog and Cowcarcass (displaced; 1.3K and 1.2K patches).

	OOBB			CAF		
	Full	ACoN	CoN	Full	ACoN	CoN
Frog	10.5%	12.1%	14.0%	14.4%	17.0%	18.4%
Frog^2	17.1%	25.1%	26.4%	22.8%	29.4%	30.9%
Cow	10.1%	14.1%	15.6%	14.5%	17.6%	18.7%
Cow^2	17.8%	27.9%	29.1%	24.0%	31.2%	32.7%

Table 1 Average cullrates for displaced models; ^{2} denotes the respective model after one level of subdivision (i.e., having four times more patches). While OOBB is our method using the bounds by Munkerberg et al. [10], CAF stands for our camera-aligned frustum.

6.2 Culling within Individual Objects

Since we perform culling on a per patch basis, we can apply our algorithm within individual models. In order to obtain meaningful cull rate measurements, we determine average cull rates by using 1K different cameras views respectively. Each view contains the entire object so that no patches are view frustum culled.

Non-displaced models: The non-displaced version of our algorithm is applied on two representative models; Killeroo and Bigguy (see Figure 8 left). Our algorithm achieves an average cull rate of 27.9% and 26.76% respectively compared to state-of-the-art back-patch culling by Loop et al. [8] that culls 38.7% and 37.8% of the patches. However, our method will cull significantly more patches with increased depth complexity (see Section 6.3), since back-patch culling algorithms cannot take advantage of inter object/patch occlusions. Note that the per patch computational cost of back-patch culling (for dynamic models) is an order of magnitude higher than ours. The effectiveness of our scheme also increaces as patch size is decreased. For instance, if we subdivide (a 1-4 split) Killeroo and Bigguy patches (5.8K and 11.6K patches) the cull rate increases to 42.5% and 42.9% respectively.

Displaced models: To test our algorithm on models with displacement maps, we used the Monsterfrog and the Cowcarcass model (see Figure 8 right). The culling rates using different cull kernels are depicted in Table 1. As expected, the uniform bounding shape extension (Full) has the lowest cull rate. Taking the approximate cone of normals into account sinigifcantly improves the cull rate. Using the accurate cone of normals (CoN) provides an additional improvement. However, the difference between ACoN and CoN is smaller than between Full and ACoN.

Furthermore, the camera-aligned frustum (CAF) is more effective than the object-oriented bounding box (OOBB) due to its better screen space bounds. Since the cost of the respective kernels is equal, we conclude that it is always better to use the CAF.

Note that these tests used simple models with a relatively small number of patches. As shown in Table 1, an increased patch count yields a significantly higher cull percentage. This results from better bounds on the displacement scalars and smaller patch sizes (small patches are more likely to be occluded).

6.3 General Culling

Culling for scenes: In Section 6.2 we considered culling within individual objects. However, more realistic applications involve scenes with multiple objects consisting of both triangle and patch meshes. Two simple example scenes are shown in Figure 1 (the ACoN kernel is used for displaced models). The first scene contains 27K patches and we achieve cull rates of 64.2% and 30.4% for the views shown in Figures 1(a) and 1(b) respectively. Rendering is speed up by a factor of 2.1 and 1.2, respectively (using a tess factor of 16). As expected, the higher the depth complexity, the more patches can be culled. Our method also benefits from triangle mesh occluders as shown in our second test scene (5.5K patches and 156 triangles). We achieve cull rates of 70.6% and 30.5% for the views shown in Figures 1(c) and 1(d). Render time is reduced by a factor of 2.6 and 1.3 (using a tess factor of 32). This demonstrates our algorithm's viability for game levels containing animated characters rendered by hardware tessellation within triangle mesh environments. In such a scenario per object culling methods would not be suitable since computing the bounding box geometry of an object on-the-fly is costly and ineffective (looping over all control points is required). Hierarchical approaches (e.g., clustering patches) would be also inappropriate since hierarchy updates create significant overhead.

Payoff: In real applications the efficiency of a culling algorithm will determine when culling becomes beneficial. This depends on patch evaluation cost, scene composition, and viewing position. By quantifying patch evaluation cost, we can provide a general statement (i.e., independent of a specific scene composition) about the cull rate needed to amortize the cost of our method. For this analysis we use relatively inexpensive bicubic

Bézier patches; more expensive to evaluate patches will benefit even more from culling.

Let RT represent the time it takes to render all patches assuming no culling. Let RT_0 represent the time it takes to render all patches with tess factor set to zero; this would be the draw time if all patches were culled. Finally, let CT represent the time it takes to perform the culling tests. We do not include the cost of Hi-Z map generation since it is constant and becomes negligible for a moderately large number of patches. For a cull rate of x, our culling pipeline (see Section 3) will have the following patch related costs: a draw call that consists of rendering the non-culled patches ($(1-x)\cdot RT$) including the culled patches with tess factor set to zero ($x \cdot RT_0$); plus the cull test time (CT); plus the cost of rendering all patches with a tess factor of zero (RT_0) since patch rendering is applied in two passes. Note that the cost of newly-visible patches is accounted for in the RT term. In order to determine the break even point (i.e., when culling becomes a win) we compare the render time for our culling method (LHS) to that of rendering without culling (RHS) in the following equation:

$$(1-x)\cdot RT + x\cdot RT_0 + CT + RT_0 = RT$$

Solving for x provides the break even point:

$$x = \frac{RT_0 + CT}{RT - RT_0}$$

Since we can measure RT at a given tess factor as well as RT_0, and CT for bicubic patches, we can create a graph for the culling break even points (see Figure 9 left). The graph plots the resulting cull rate x as a function of tess factor using the ACoN cull kernel. Graphs for the NoDisp and Full kernel are almost equal to that of ACoN. The CoN kernel, however, does not pay off until a tess factor of 6 with a cull rate of over 60%. Since the cull rate of CoN is only slightly better, but its costs are an order of magnitude higher than ACoN, the ACoN kernels generally provides better performance. An application for CoN would be static (non-animated) objects where the cone of normals can be precomputed.

Temporal Coherence: In order to measure the effects of reduced temporal coherence on our algorithm, we rotate the Bigguy model around its z-axis using different speeds of rotation (see Figure 9 right). We choose rotations for this test as it is worst-case in terms of violating temporal visibility coherence. Even for large view point changes, the cull rate is only slightly affected. In our example the cull rate drops from 25.5% (no movement) to 20.4% (rotation at a speed of 10° per frame). While we expect extreme scene motion to reduce cull rates, we note that due to the conservative nature of our algorithm, no visible patches will be missed.

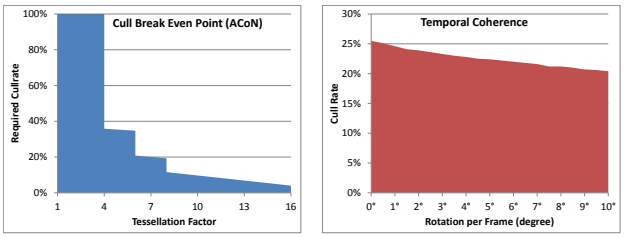


Fig. 9 Left: required cull rates to make culling beneficial for different tess factors using the ACoN kernel. Culling becomes a win above a tess factor of 4 at a cull rate of 35.8%. Right: average cull rates using the Bigguy model for different speeds of rotation around the z-axis. Even for a large temporal change, our algorithm's cull rate is only slightly affected.

Adaptive tessellation: An important advantage of the hardware tessellation pipeline is the ability to assign tess factors to individual patches dynamically. While this can reduce tessellation costs significantly, an additional constant amount of per patch computation is required to determine these tess factors. Within the context of our culling method, this means we can avoid tess factor computations for culled patches. However, since adaptive tessellation tends to assign smaller tess factors to potentially culled patches, there is less room for saving tessellation costs. In order to evaluate the culling efficiency of our algorithm when using adaptive tessellation we use a simple camera distance based tess factor estimate (similar to the DetailTessellation11 sample of the DirectX SDK). For the scene and camera setup of Figure 1(a) we achieve roughly the same visual quality at render times of 3.8 ms (w/o culling) and 2.6 ms (w/ culling; ACoN), respectively. Thus, our method reduces render time by about 31.6%. Please note that more sophisticated adaptive tess factor estimates (e.g., curvature based) would favor our algorithm even more due to the extra tess factor computation costs. For the same setup using the view of Figure 1(b), where we have only intra-object occlusion (cull rate of 30.4%), our culling method does not pay off; i.e., about 9% slower than w/o culling. In order to avoid such a situation where our method's overhead is not amortized by savings, we suggest disabling culling for selected models. This decision can be made for each model independently. For instance, one could compute a single adaptive tess factor per model (e.g., model centroid to eye distance); if that tess factor is less than a particular threshold (see Figure 9 left for payoff), do not apply culling. Per patch tess factors can be still assigned afterwards to non-culled patches.

Another option for adaptive tessellation is to compute a uniform tess factor for each model at runtime.

That would allow balancing costs and gains for each model individually, and provide a decision whether or not to apply culling.

7 Conclusion and Future Work

We have presented a simple, yet effective algorithm for culling occluded bicubic patches for hardware tessellation. We used bicubic Bézier patches due to their simplicity and popularity; however, our method can be applied with different types of patches such as PN-Triangles [16] or Gregory patches [7]. All that we require are methods to bound patch geometry, and first partial derivatives (to determine an approximate cone of normals).

Our results show that our culling method performs well on current hardware involving only minimal overhead. Thus, culling is effective even for simple scenes (e.g., single objects) and small tess factors (see Figure 9). In addition our patch-based culling algorithm can easily be combined with previous per object occlusion culling methods that are applied on triangle meshes. We believe that our method is ideally suited for real-time applications that leverage the advantages of hardware tessellation.

Acknowledgements We would like to thank Bay Raitt for the Bigguy and Monsterfrog model. The Killeroo model is courtesy of Headus (metamorphosis) Pty Ltd. Further, we thank the Blender Foundation for the open movie project Sintel which is the source of the Cowcarcass model.

References

1. Bittner J, Wimmer M, Piringer H, Purgathofer W (2004) Coherent hierarchical culling: Hardware occlusion queries made useful. In: Computer Graphics Forum, Wiley Online Library, vol 23, pp 615–624
2. Cohen-Or D, Chrysanthou Y, Silva C, Durand F (2003) A survey of visibility for walkthrough applications. IEEE Transactions on Visualization and Computer Graphics pp 412–431
3. Engelhardt T, Dachsbacher C (2009) Granular visibility queries on the GPU. In: Proceedings of the 2009 symposium on Interactive 3D graphics and games, ACM, pp 161–167
4. Greene N, Kass M, Miller G (1993) Hierarchical z-buffer visibility. In: Proceedings of the 20th annual conference on Computer graphics and interactive techniques, ACM, pp 231–238
5. Guthe M, Balázs A, Klein R (2006) Near optimal hierarchical culling: Performance driven use of hardware occlusion queries. In: Eurographics Symposium on Rendering 2006
6. Hasselgren J, Munkberg J, Akenine-Möller T (2009) Automatic pre-tessellation culling. ACM Transactions on Graphics (TOG) 28(2):19
7. Loop C, Schaefer S, Ni T, Castano I (2009) Approximating subdivision surfaces with gregory patches for hardware tessellation. ACM Transactions on Graphics (TOG) 28(5):1–9
8. Loop C, Nießner M, Eisenacher C (2011) Effective Back-Patch Culling for Hardware Tessellation. In: Proceedings of VMV 2011: Vision, Modeling & Visualization, Berlin, Germany, pp 263–268
9. Mattausch O, Bittner J, Wimmer M (2008) CHC++: Coherent hierarchical culling revisited. In: Computer Graphics Forum, Wiley Online Library, vol 27, pp 221–230
10. Munkberg J, Hasselgren J, Toth R, Akenine-Möller T (2010) Efficient bounding of displaced Bézier patches. In: Proceedings of the Conference on High Performance Graphics, Eurographics Association, pp 153–162
11. Nießner M, Loop C, Meyer M, Derose T (2012) Feature-adaptive GPU rendering of Catmull-Clark subdivision surfaces. ACM Transactions on Graphics (TOG) 31(1):6
12. Sederberg T, Meyers R (1988) Loop detection in surface patch intersections. Computer Aided Geometric Design 5(2):161–171
13. Sekulic D (2004) Efficient occlusion culling. GPU Gems pp 487–503
14. Shirmun L, Abi-Ezzi S (1993) The cone of normals technique for fast processing of curved patches. In: Computer Graphics Forum, Wiley Online Library, vol 12, pp 261–272
15. Shopf J, Barczak J, Oat C, Tatarchuk N (2008) March of the Froblins: simulation and rendering massive crowds of intelligent and detailed creatures on GPU. In: ACM SIGGRAPH 2008 classes, ACM, pp 52–101
16. Vlachos A, Peters J, Boyd C, Mitchell J (2001) Curved PN triangles. In: Proceedings of the 2001 symposium on Interactive 3D graphics, ACM, pp 159–166